B-Trees

Thomas Schwarz SJ

B-Trees

- Standard data structure for Key-Value Stores
  - Stores records, composed of key and value
    - Assumes that keys are ordered
  - Implements CRUD: Create, Read, Update, Delete given a key
  - Implements range queries: Recover all records with an id in a certain range

B-Trees

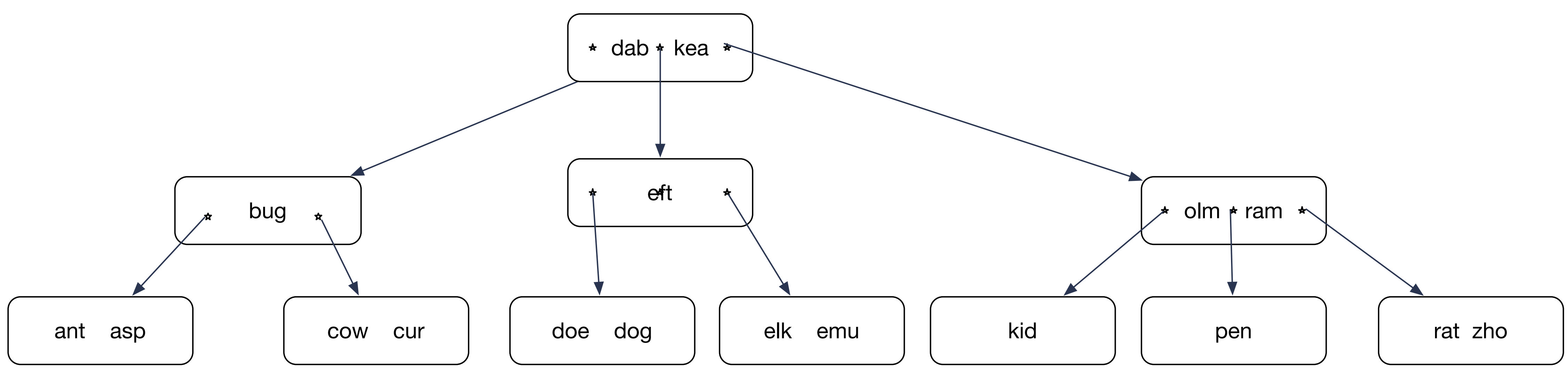
- B-Trees proper:
  - Stores records in pages in memory
- B+-Tree:
  - Variant that stores data in pages in storage

B-Trees

- B-trees: In memory data structure for CRUD and range queries
  - Balanced Tree
  - Each node can have between *d* and 2*d* keys with the exception of the root
  - Each node consists of a sequence of node pointer, key, node pointer, key, …, key, node pointer
  - Tree is ordered.
    - All keys in a child are between the keys adjacent to the node pointer

B-Trees

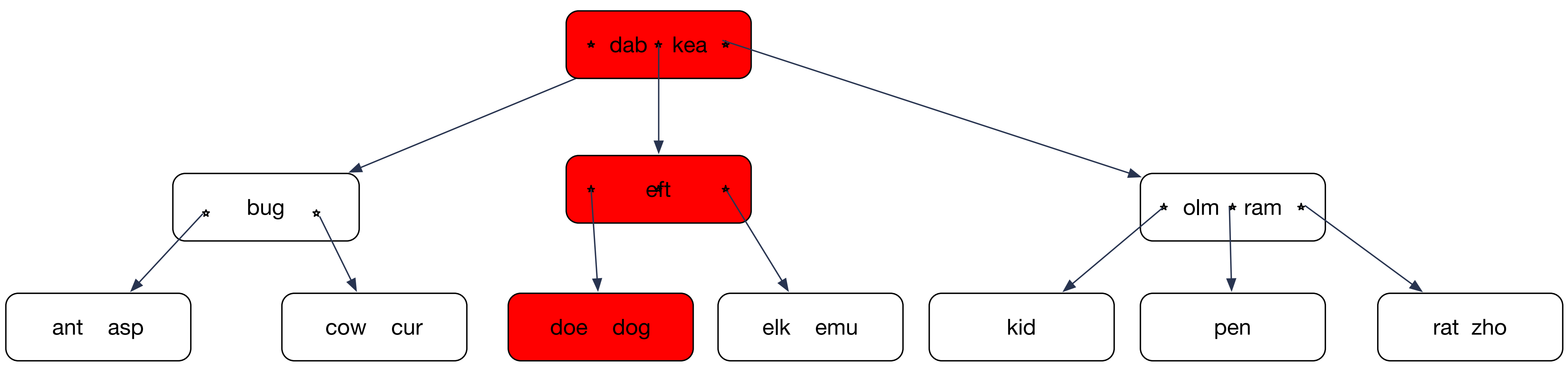
- Example: 2-3 tree: Each node has two or three children



B-Trees

- Read dog:
  - Load root, determine location of dog in relation to the keys
  - Follow middle pointer
  - Follow pointer to the left
  - Find “dog”

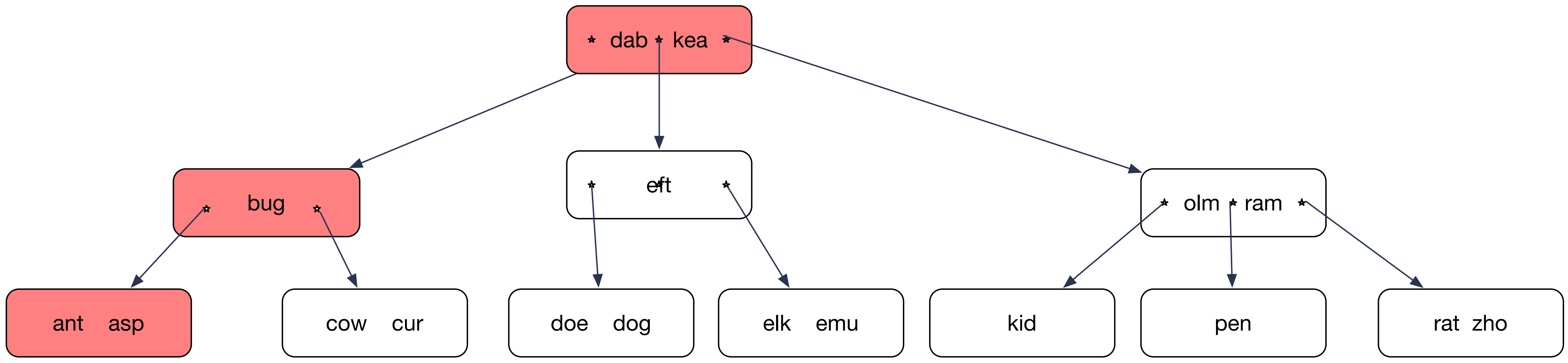
B-Trees



dab kea
bug
eft
olm ram
ant asp
cow cur
doe dog
elk emu
kid
pen
rat zho

B-Trees

- Search for “auk” :



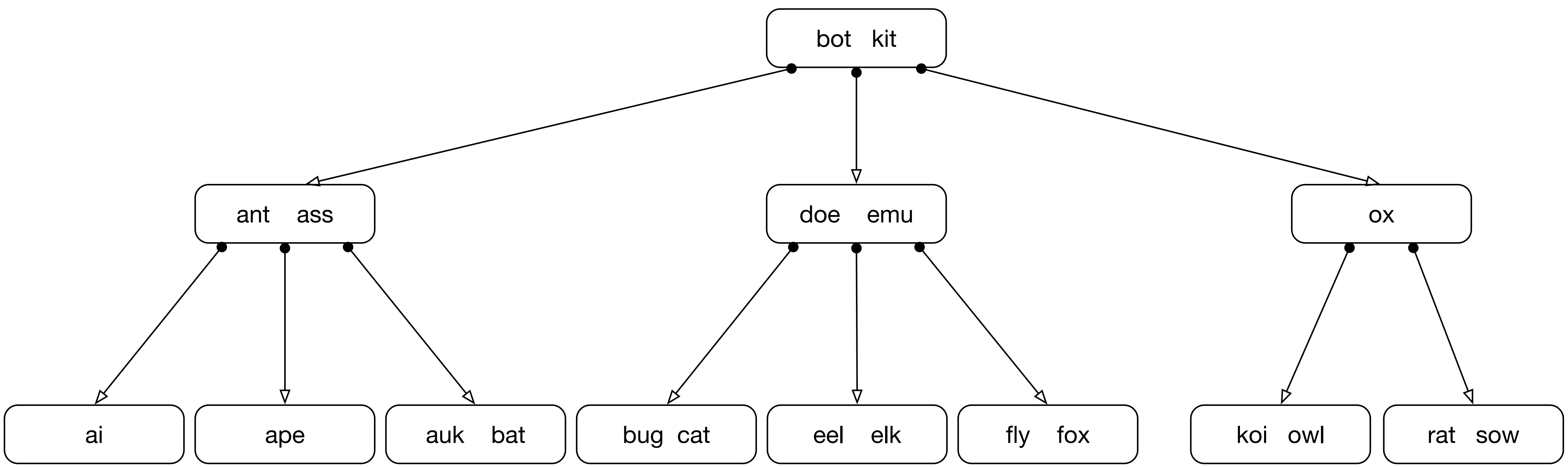
B-Trees

- In-order traversal
  - Recursive operation
  - If node contains $l_0, k_1, l_1, k_2, l_2, \ldots, l_{d-1}, k_d, l_d$
    - With links l_i and keys k_i:

```
for i in range(d):
     in_order_traversal(l[i])
     emit(k[i])
in_order_traversal(l[d])
```

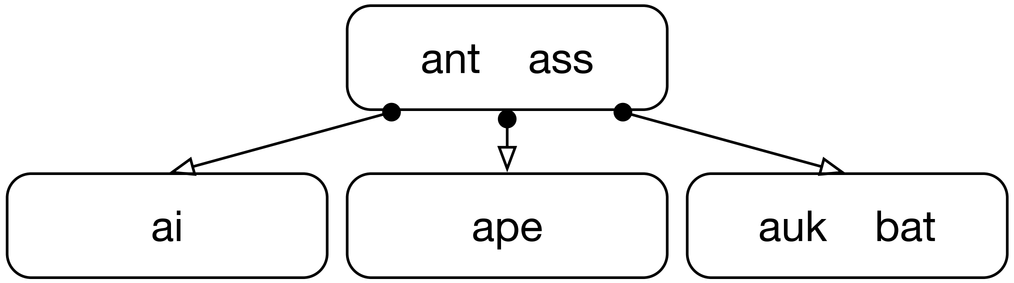
B-Trees

- Example:
  - in_order of

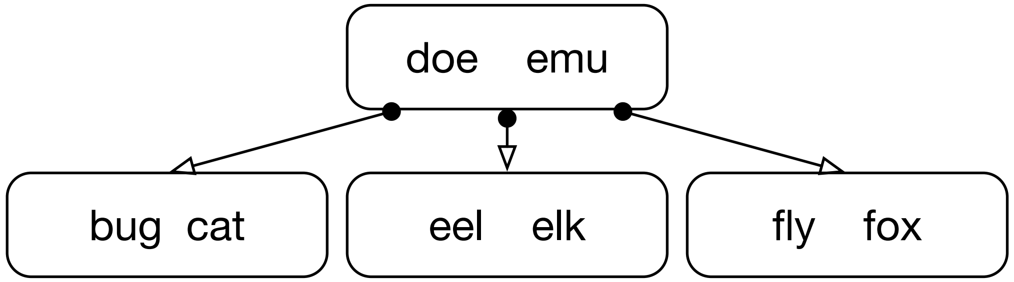


B-Trees

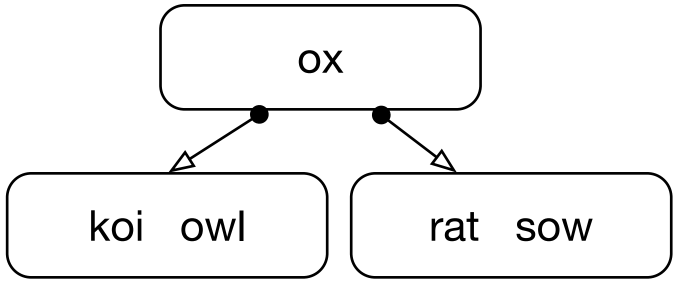
- in-order of



- 'bot'
- in-order of

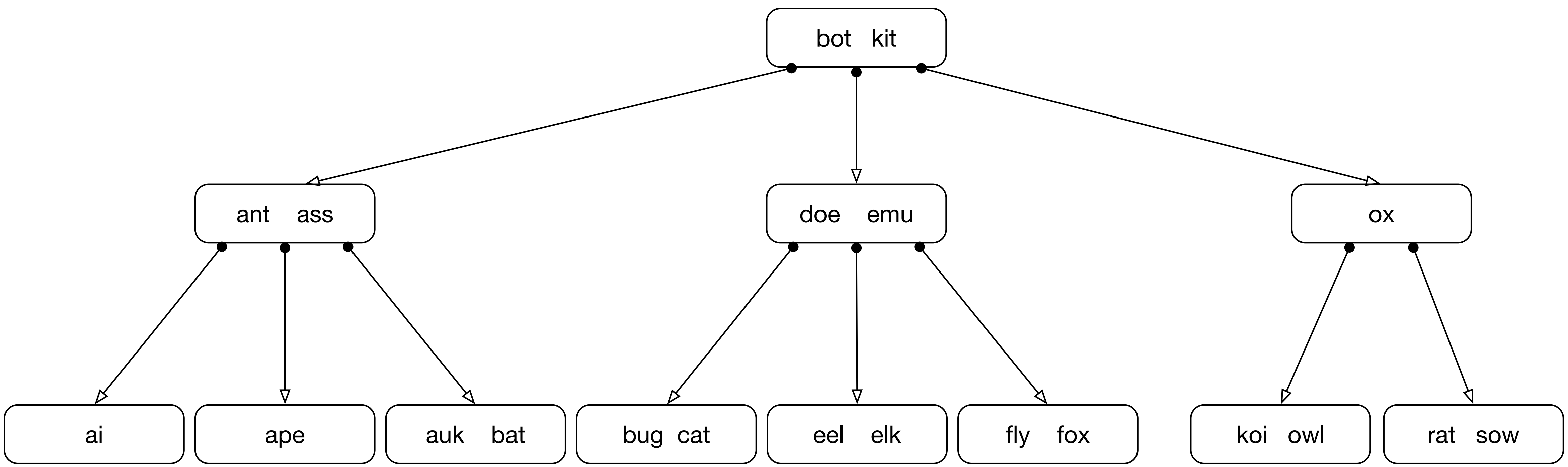


- 'kit'
- in-order of



B-Trees

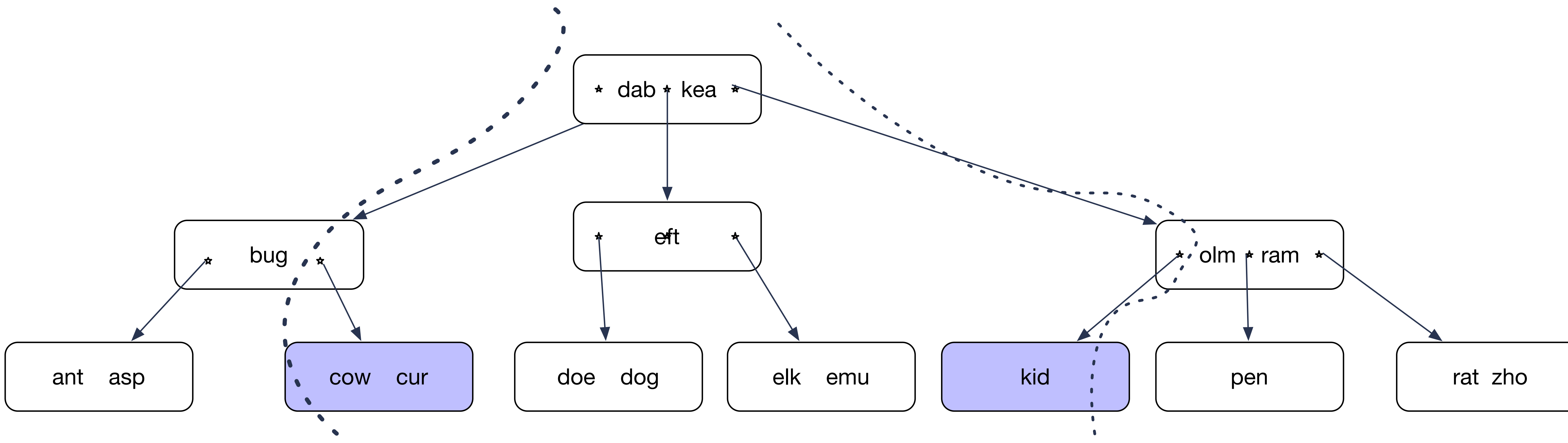
- in-order traversal



- ai ant ape ass auk bat bot bug cat doe eel elk emu fly fox kit koi owl ox rat sow

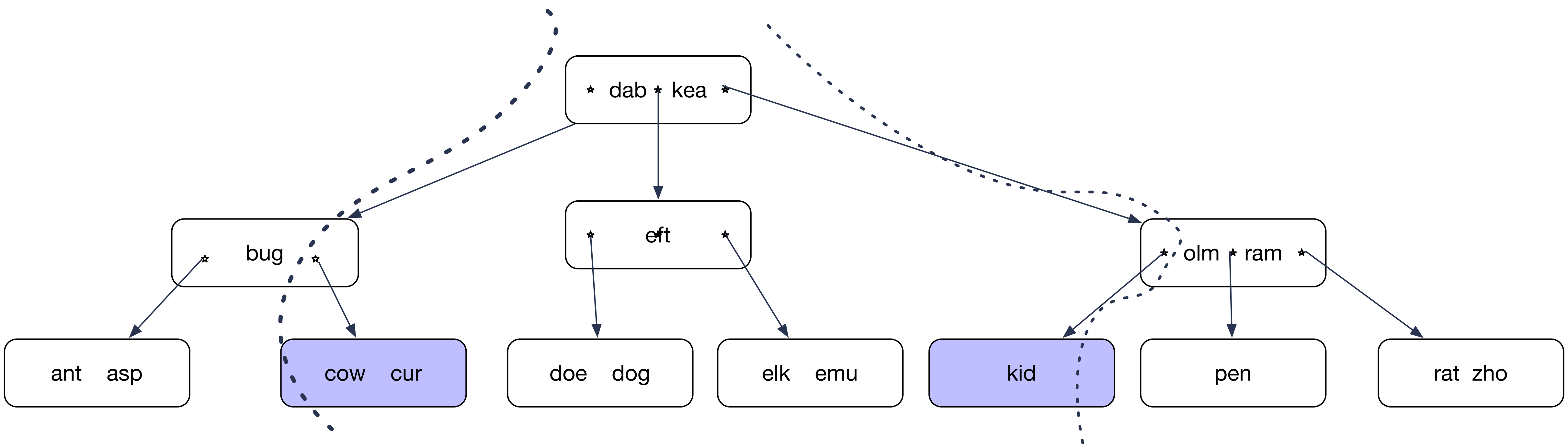
B-Trees

- Range Query c - l
  - Determine location of c and l



B-Trees

- Recursively enumerate all nodes between the lines starting with root



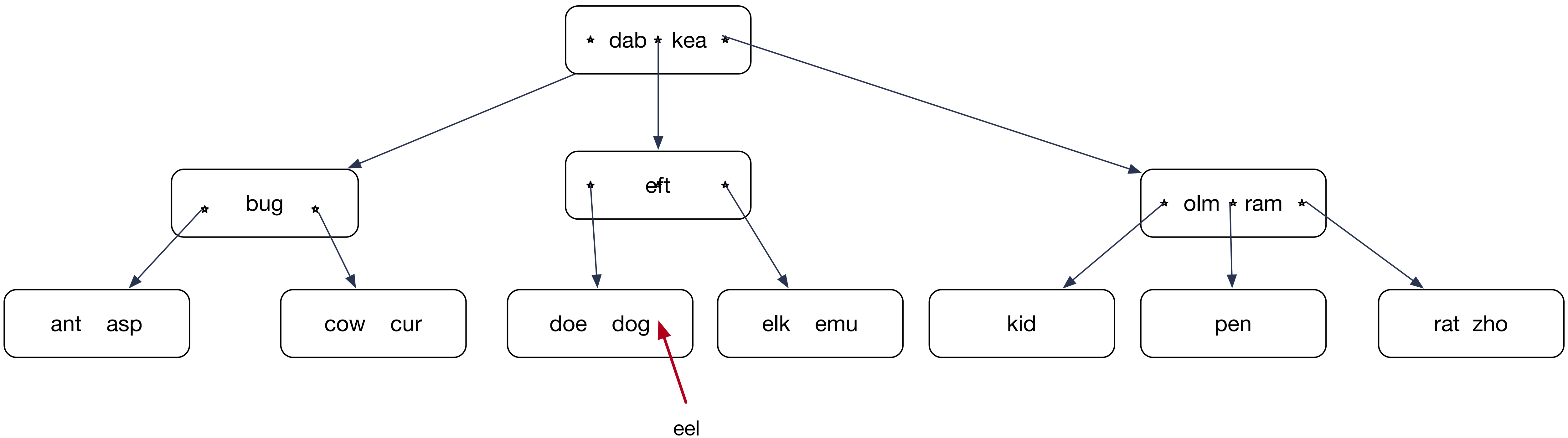
B-trees

- Capacity: With *l* levels, minimum of $1 + 2 + 2^2 + \ldots + 2^l$ records:
  - $1(2^{l+1} - 1)$ keys
- Maximum of $1 + 3 + 3^2 + \ldots + 3^l$ records
  - $\frac{2}{2}(3^{l+1} - 1)$ keys

B-trees

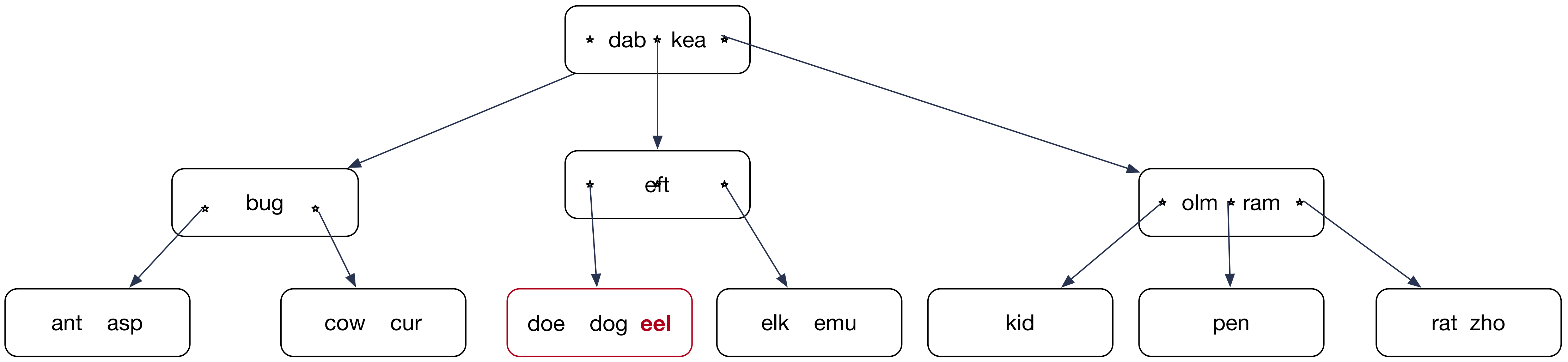
- Inserts:
  - Determine where the key should be located in a leaf
  - Insert into leaf node
  - Leaf node can now have too many keys
  - Take middle key and elevate it to the next higher level
  - Which can cause more “splits”

B-trees



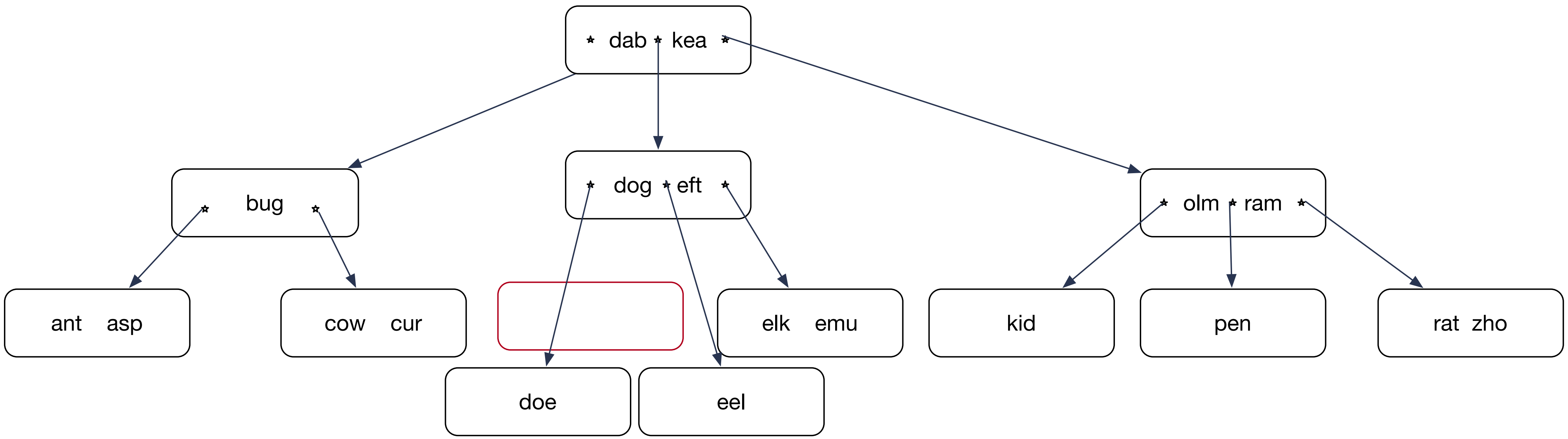
dab
kea
bug
eft
olm
ram
ant asp
cow cur
doe dog
elk emu
kid
pen
rat zho
eel

B-trees



dab
kea
bug
eft
olm
ram
ant asp
cow cur
doe dog eel
elk emu
kid
pen
rat zho

B-trees



dab
kea
bug
dog
eft
olm
ram
ant
asp
cow
cur
elk
emu
kid
pen
rat
zho
doe
eel

B-trees

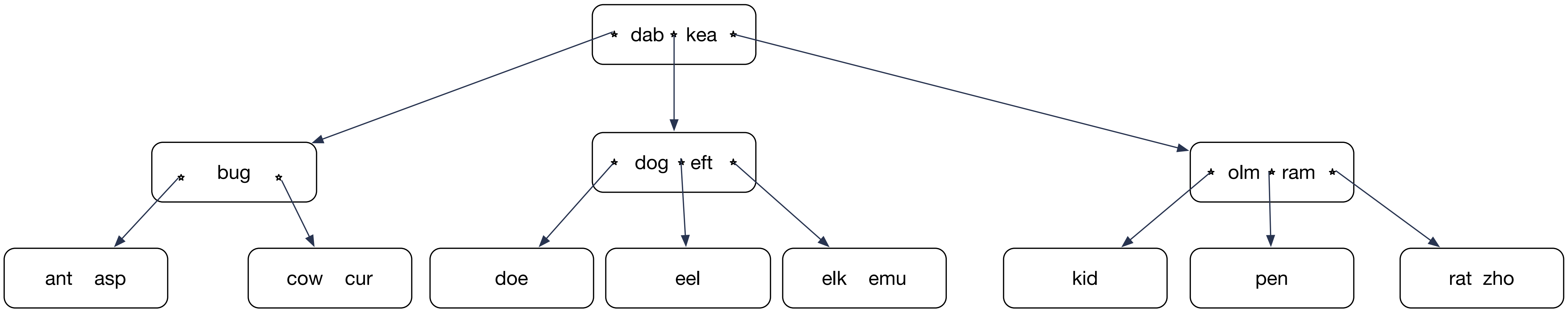
- Insert: Lock all nodes from root on down so that only one process can operate on the nodes
- Tree only grows a new level by splitting the root

B-Trees

- Using only splits leads to skinny trees
  - Better to make use of potential room in adjacent nodes
  - Insert “ewe”.
    - Node elk-emu only has one true neighbor.
      - Node kid does not count, it is a cousin, not a sibling

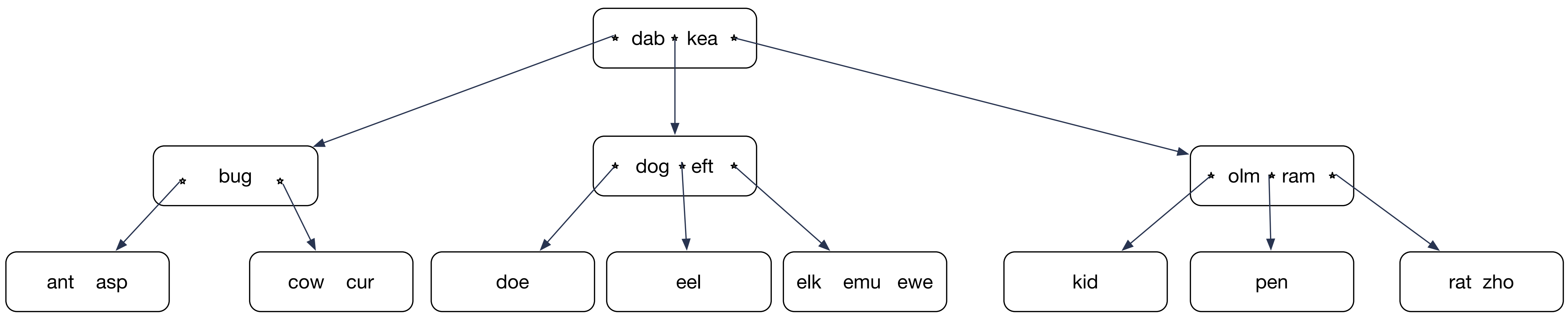
B-tree

- Insert ewe into



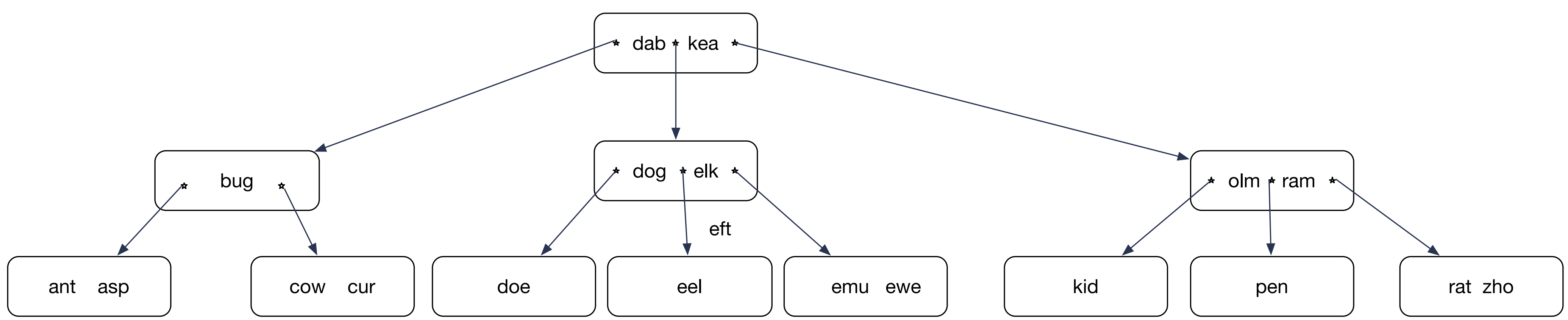
B-tree

- Insert ewe



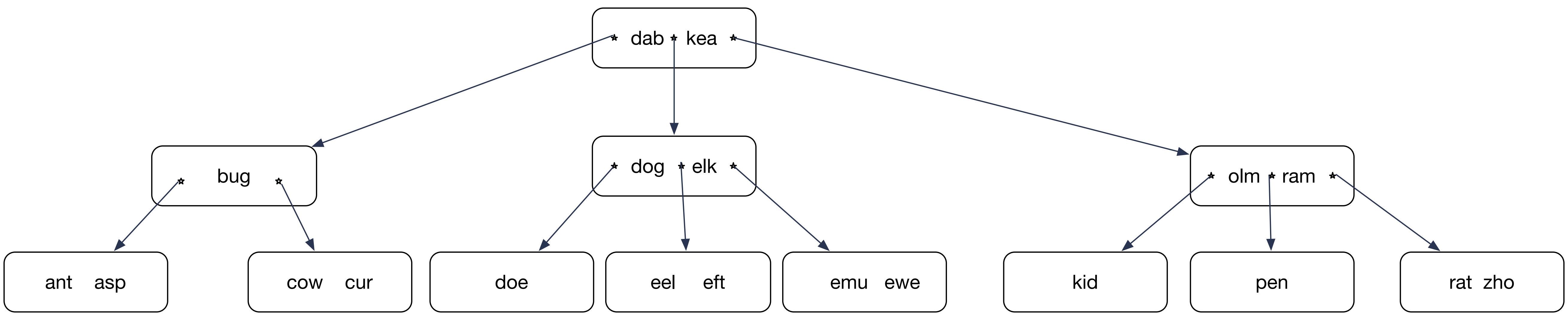
B-tree

- Promote elk. elk is guaranteed to come right after eft.
- Demote eft



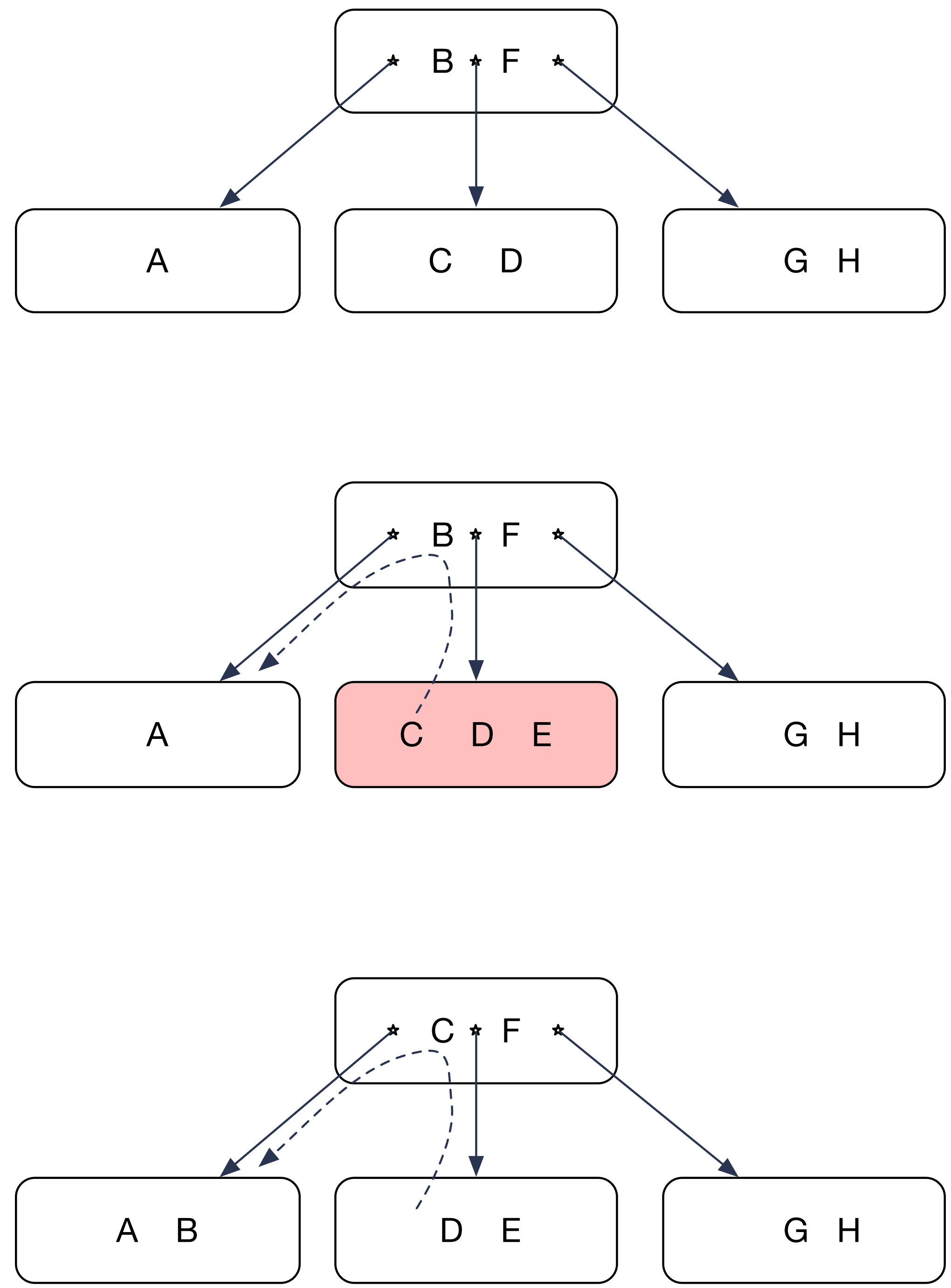
B-tree

- Insert eft into the leaf node

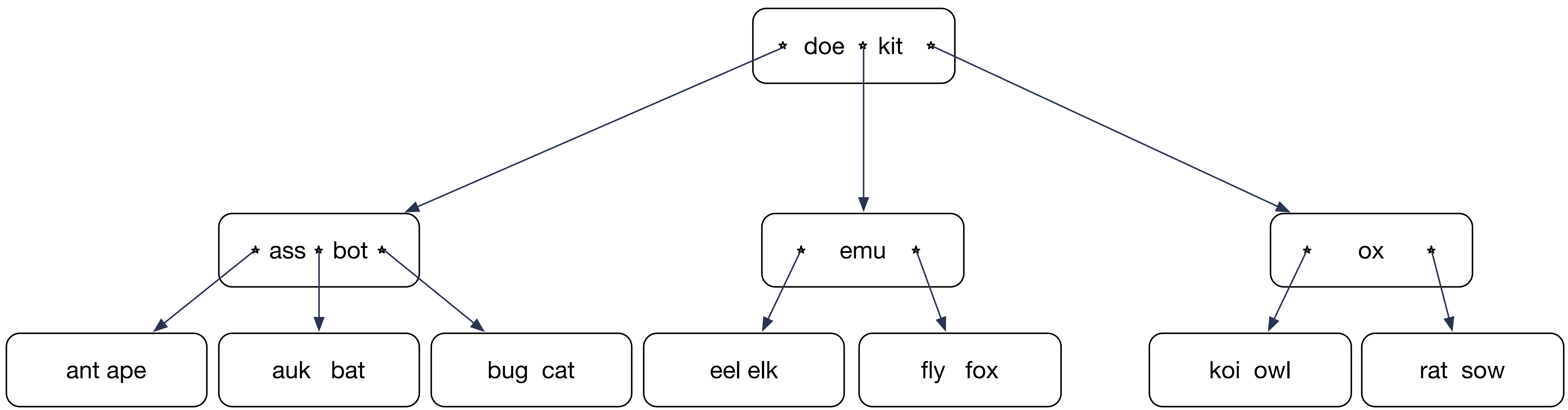


B-tree

- Left rotate
  - Overflowing node has a sibling to the left with space
  - Move left-most key up
  - Lower left-most key

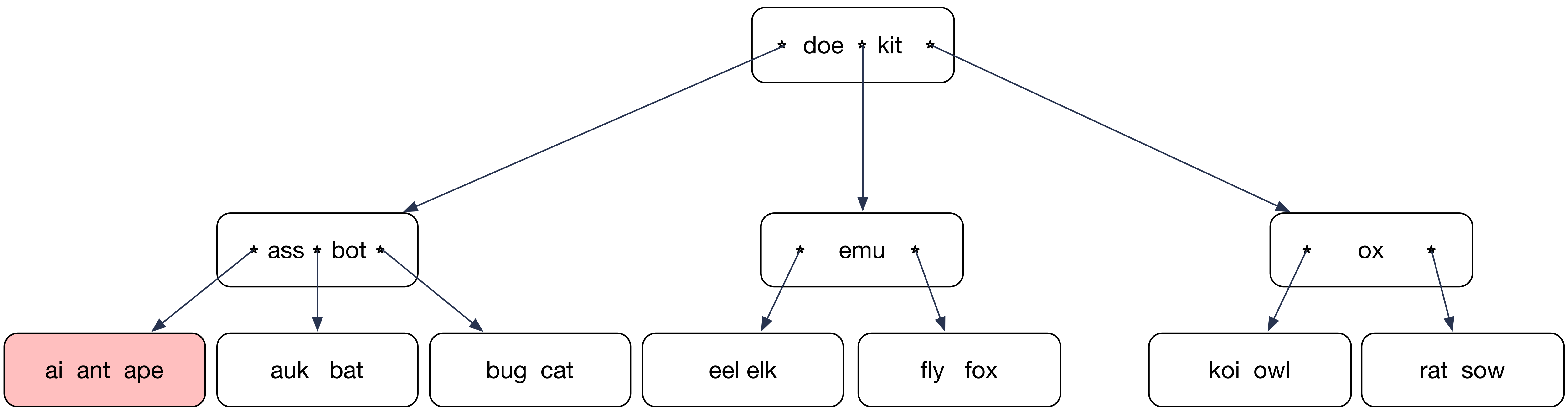


B-tree



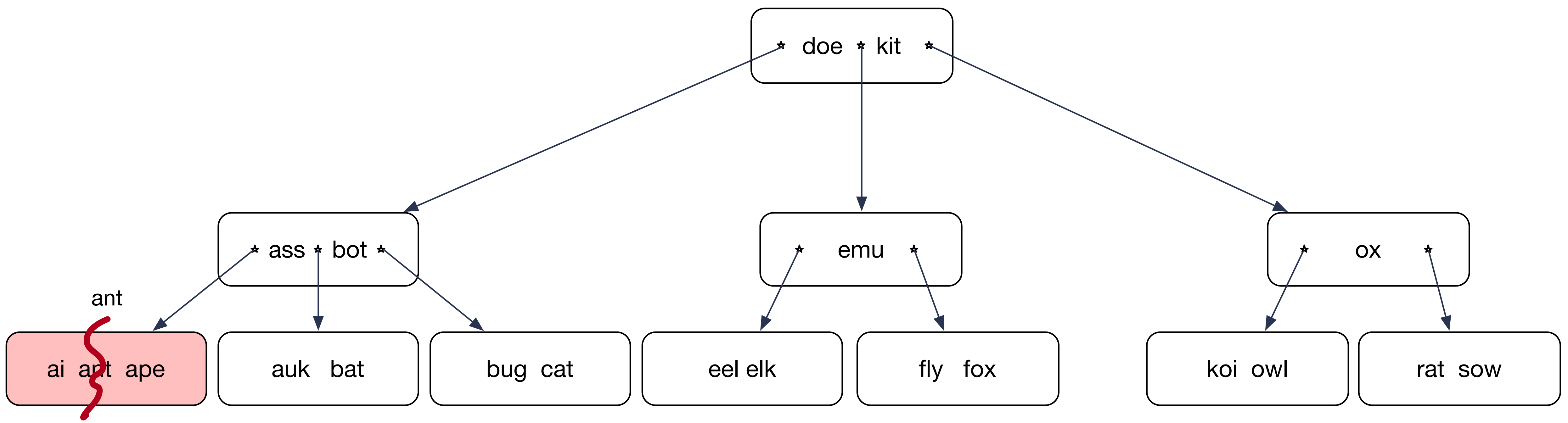
Now insert "ai"

B-tree



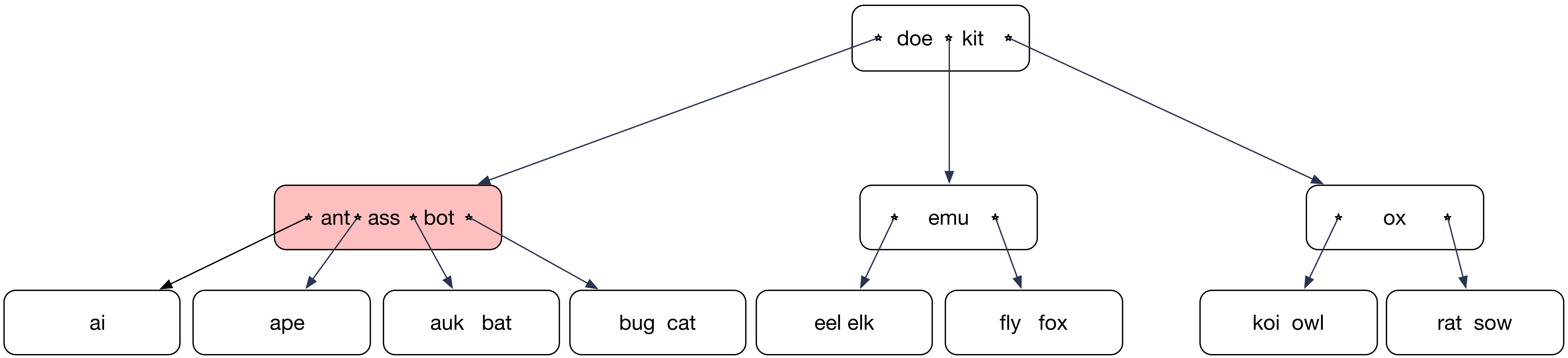
Insert creates an overflowing node
Only one neighboring sibling, but that one is full
Split!

B-tree



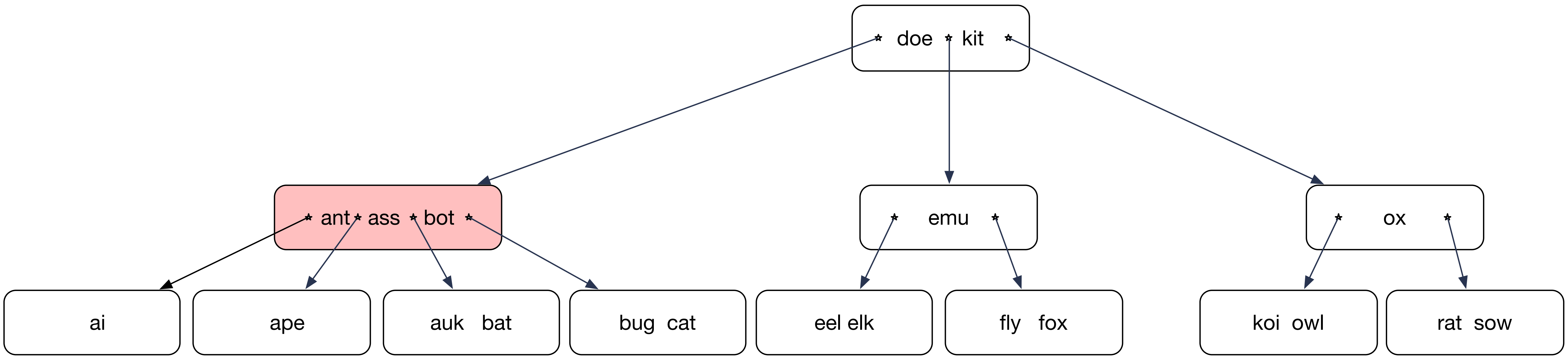
Middle key moves up

B-tree



Unfortunately, this gives another overflow
But this node has a right sibling not at full capacity

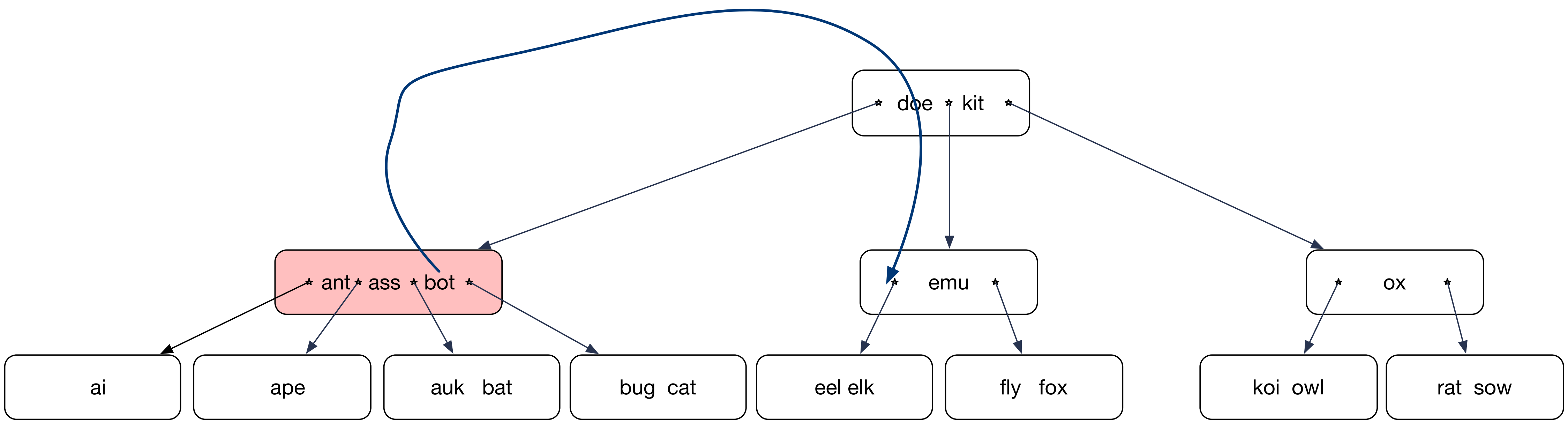
B-tree



Right rotate:
Move “bot” up
Move “doe” down
Reattach nodes

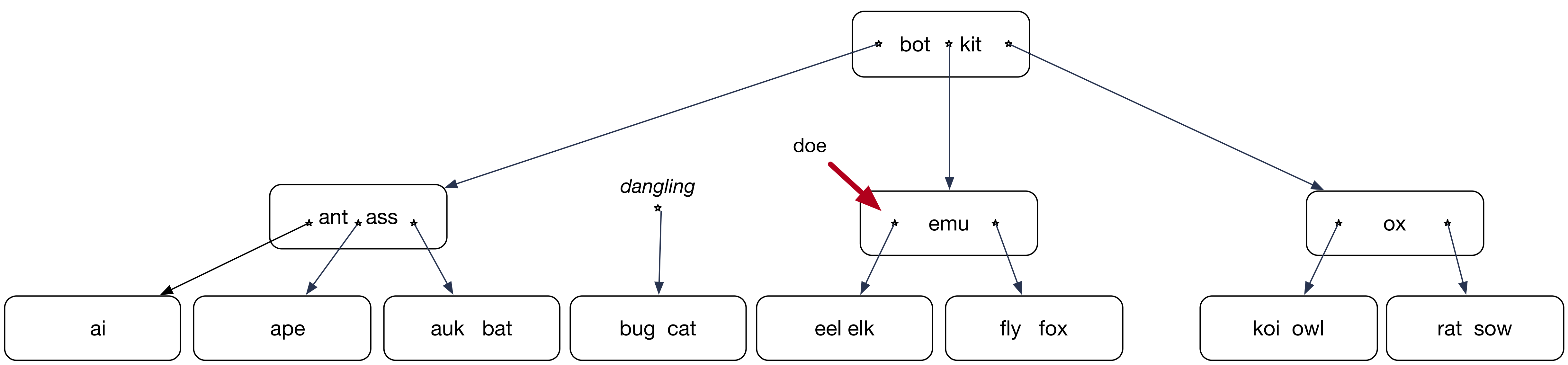
'bug', 'cat' are bigger than 'bot and smaller than 'doe'

B-tree



Move "bot" up
Move "doe" down
Reattach the dangling node

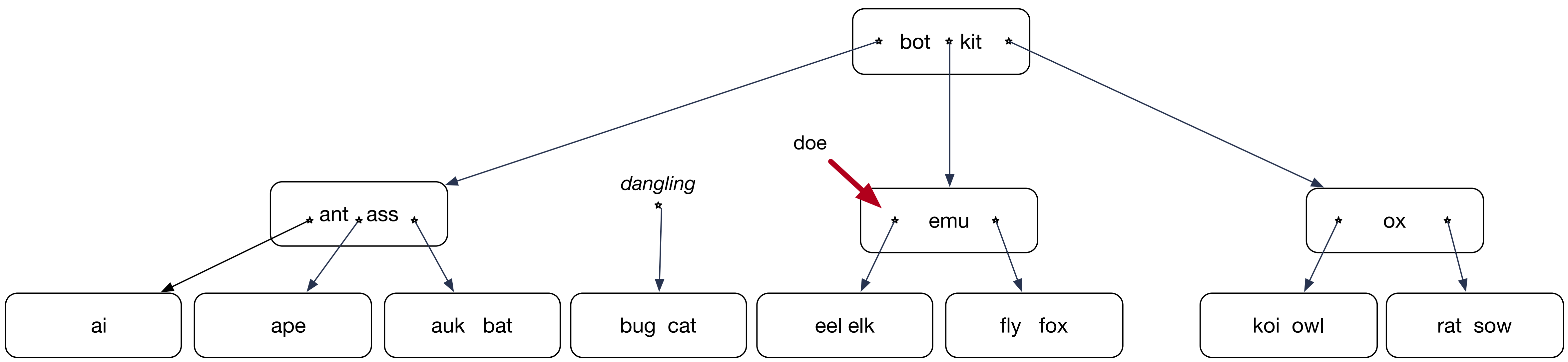
B-tree



“bot” had moved up and replaced doe

The “emu” node needs to receive one key and one pointer

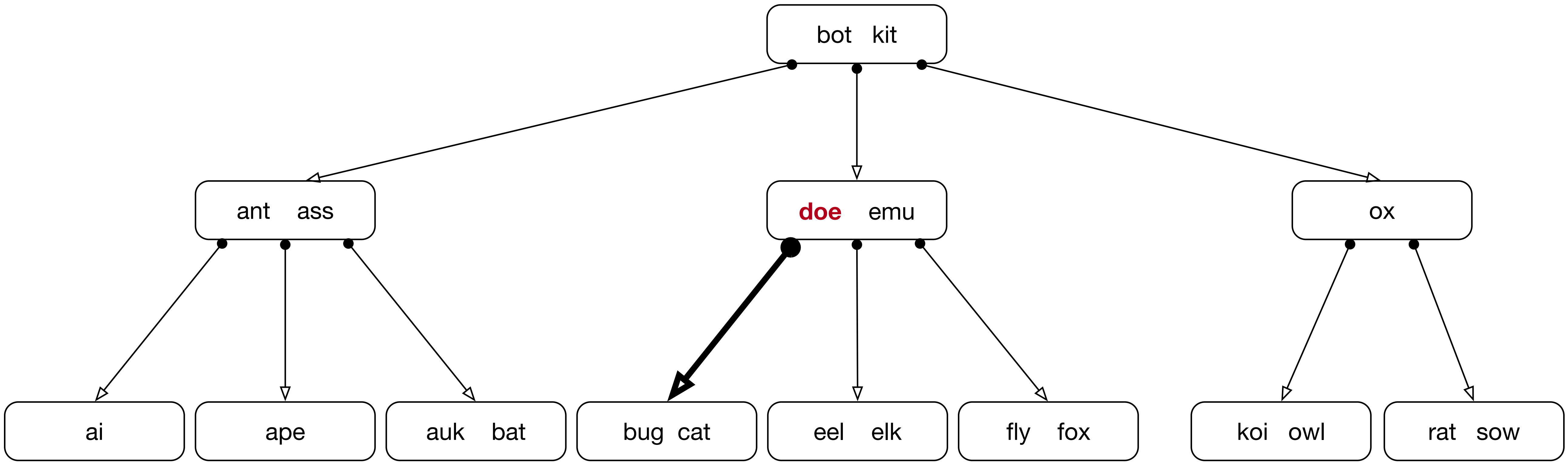
B-tree



bot kit
doe
dangling
ant ass
emu
ox
ai
ape
auk bat
bug cat
eel elk
fly fox
koi owl
rat sow

B-Tree

- When 'doe' becomes part of the node, a slot for a new left-most node opens up



- 'bug' 'cat' are larger than 'bot' and smaller than 'doe'

B-tree

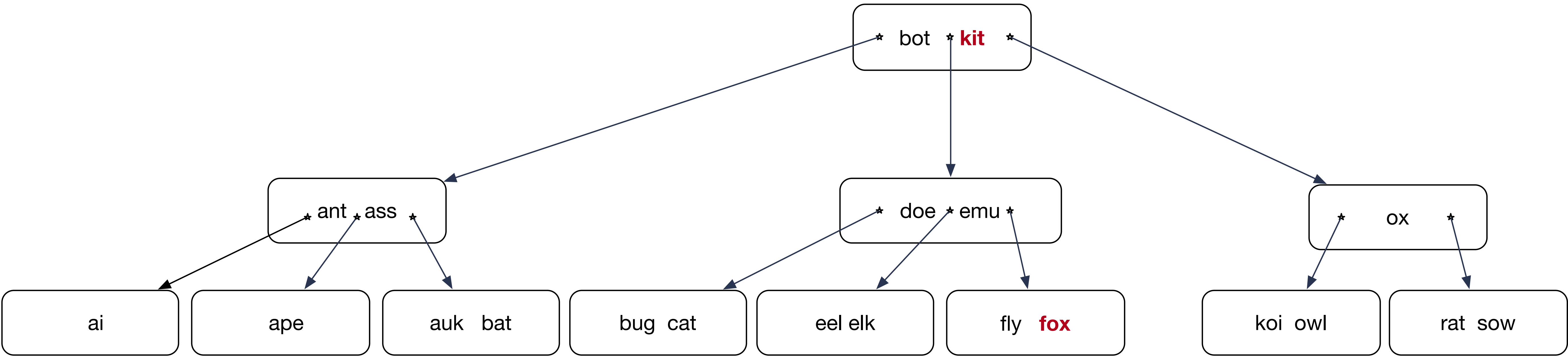
- Deletes
  - Usually restructuring not done because there is no need
  - Underflowing nodes will fill up with new inserts

B-tree

- Implementing deletion anyway:
  - Can only remove keys from leaves
  - If a delete causes an underflow, try a rotate into the underflowing node
  - If this is not possible, then merge with a sibling
    - A merge is the opposite of a split
  - This can create an underflow in the parent node
    - Again, first try rotate, then do a merge

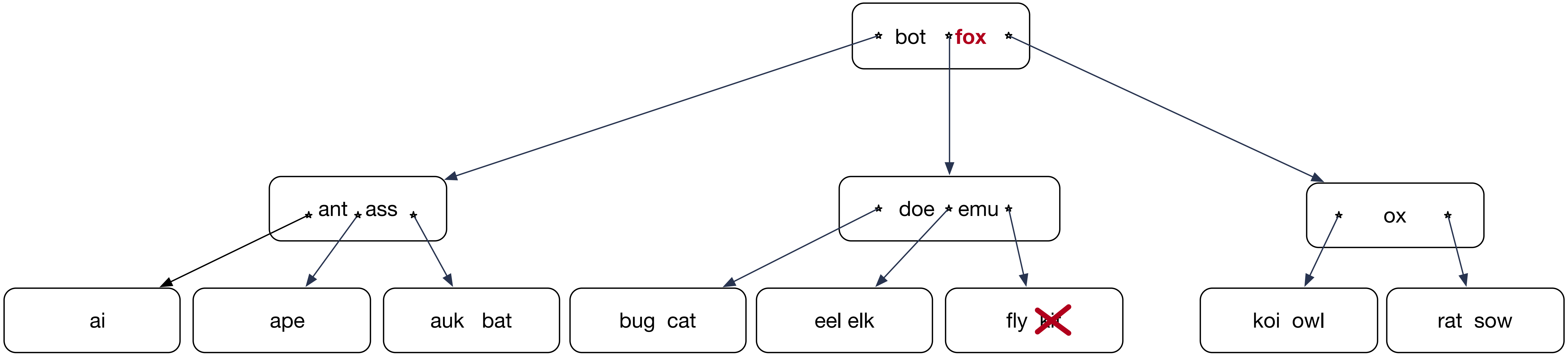
B-tree

Delete “kit”



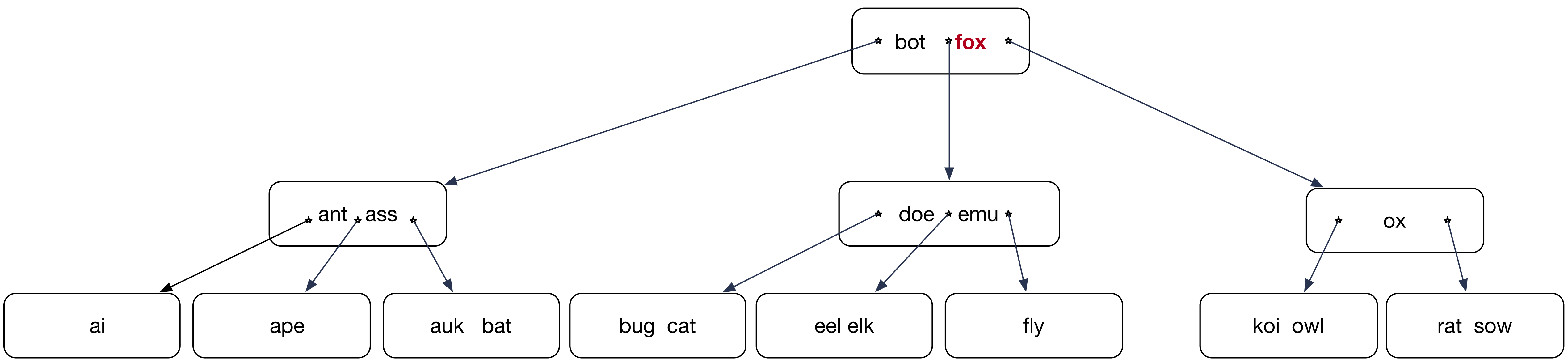
Delete “kit”
“kit” is in an interior node.
Exchange it with the key in the leave immediately before
“fox”

B-tree



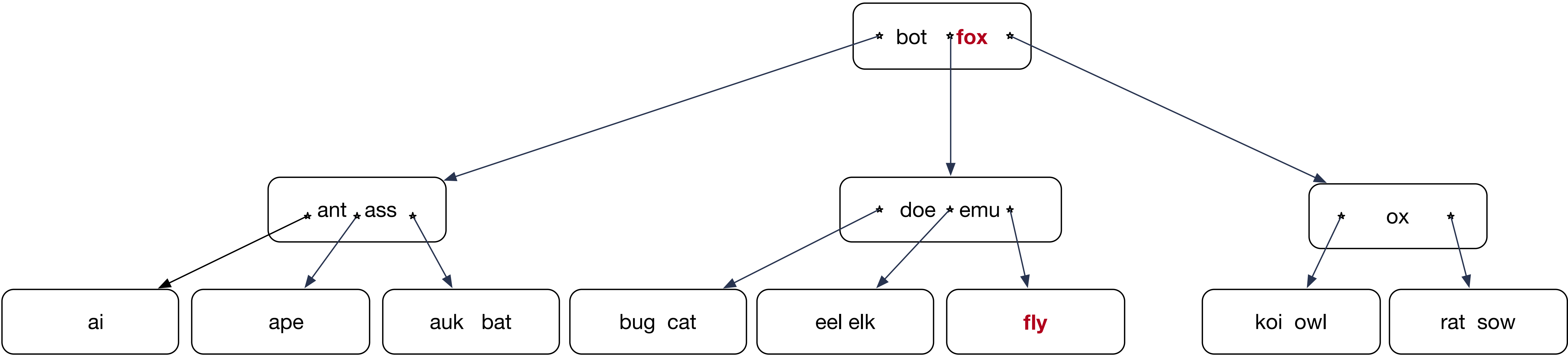
After interchanging "fox" and "kit", can delete "kit"

B-tree



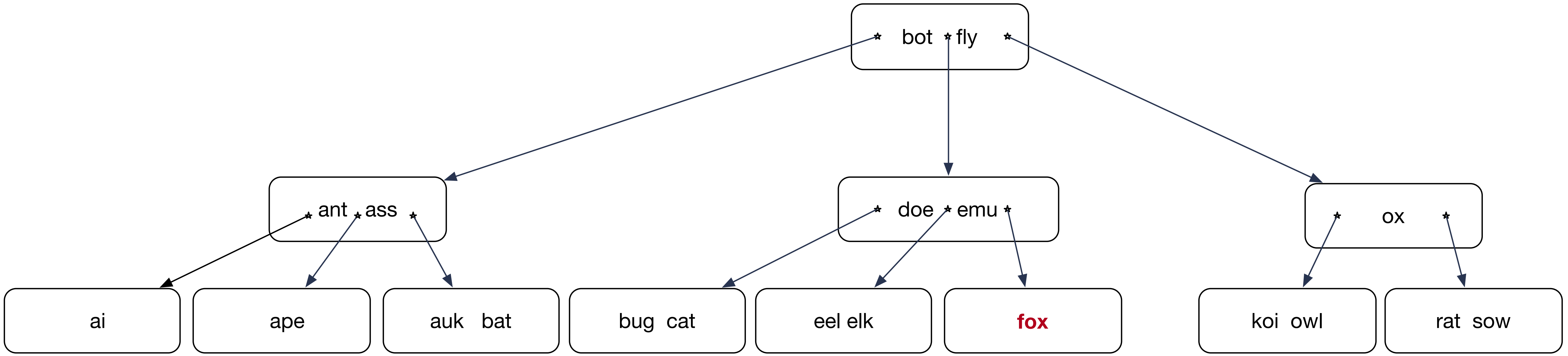
Now delete “fox”

B-tree



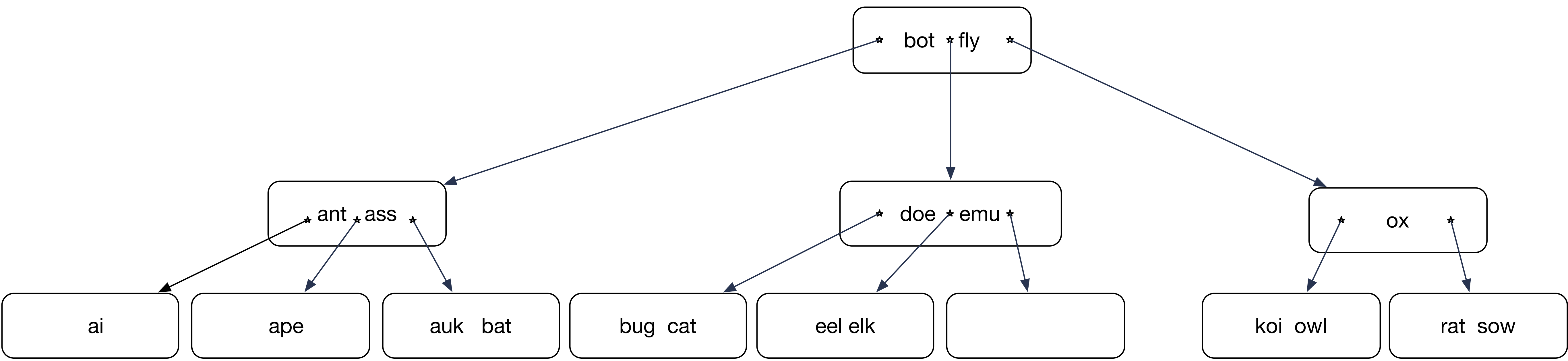
Step 1: Find the key. If it is not in a leaf
Step 2: Determine the key just before it, necessarily in a leaf
Step 3: Interchange the two keys

B-tree



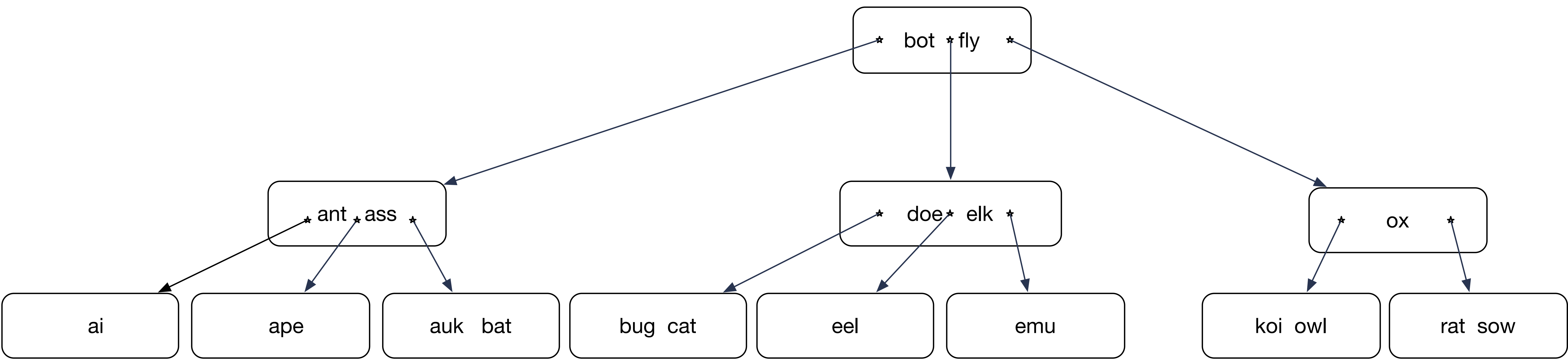
Step 4: Remove the key now from a leaf

B-tree



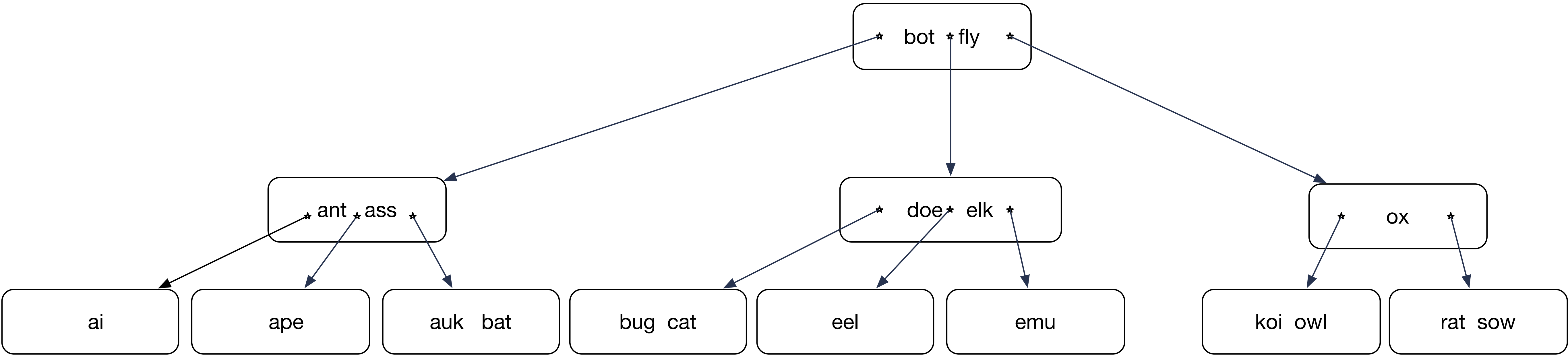
This causes an underflow
Remedy the underflow by right rotating from the sibling

B-tree



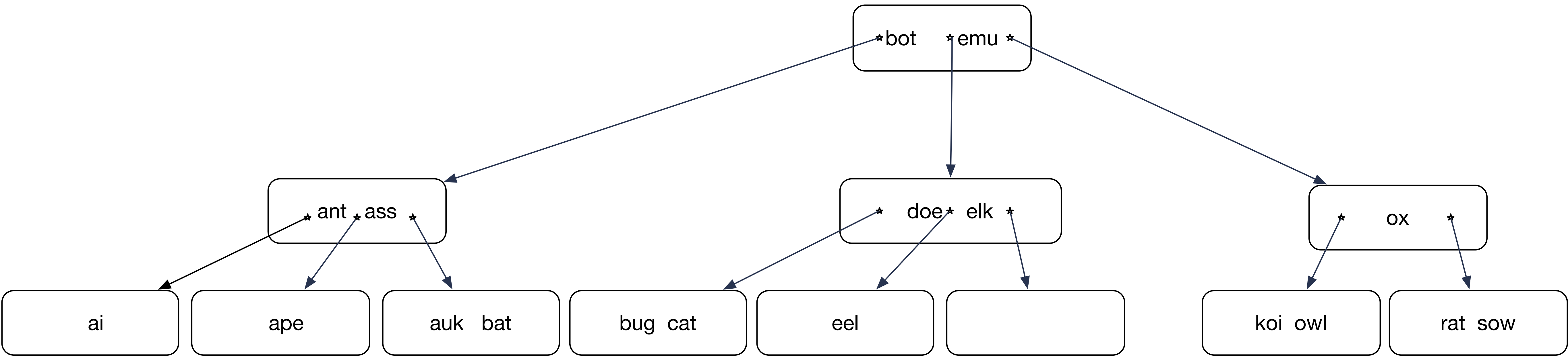
Everything is now in order

B-tree



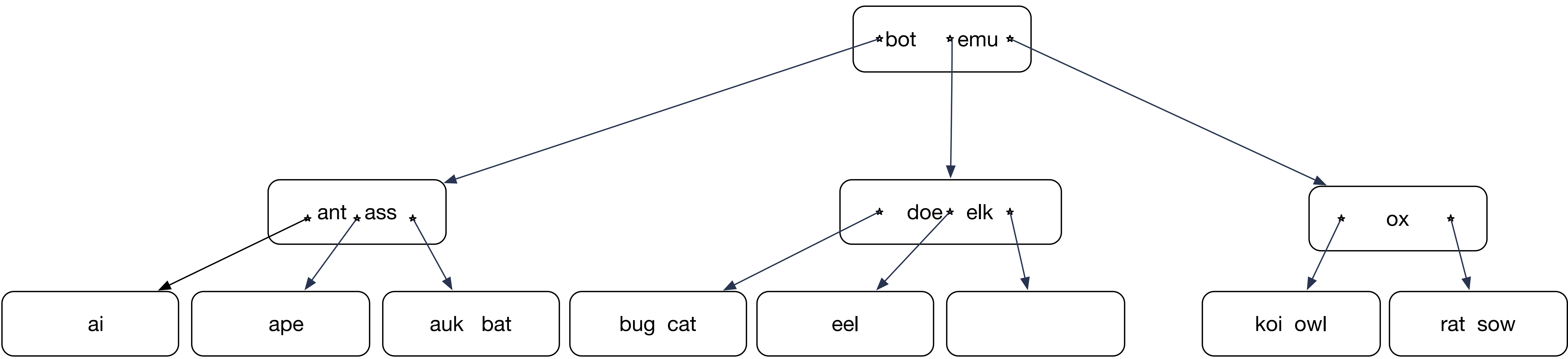
Now delete fly

B-tree



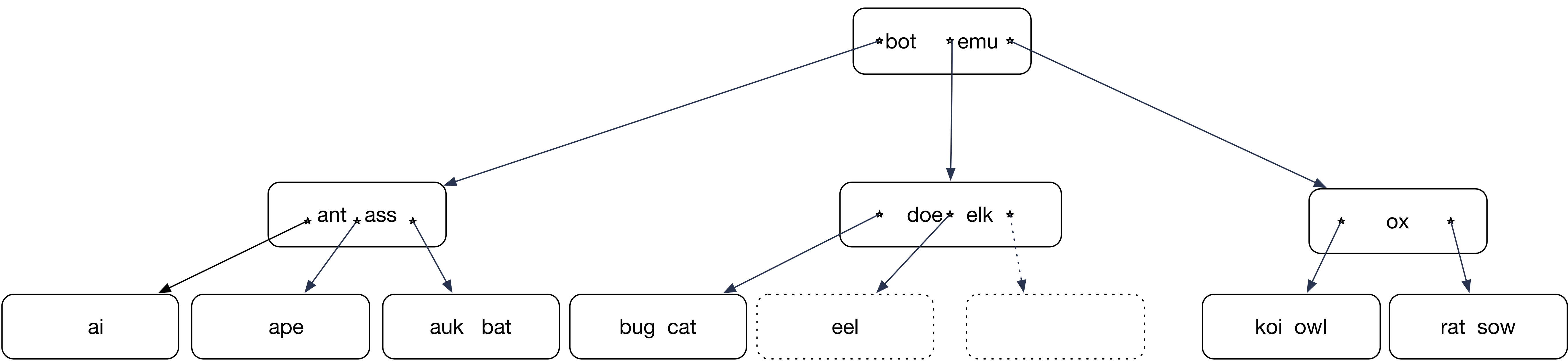
Switch “fly” with “emu”
remove “fly” from the leaf
Again: underflow

B-tree



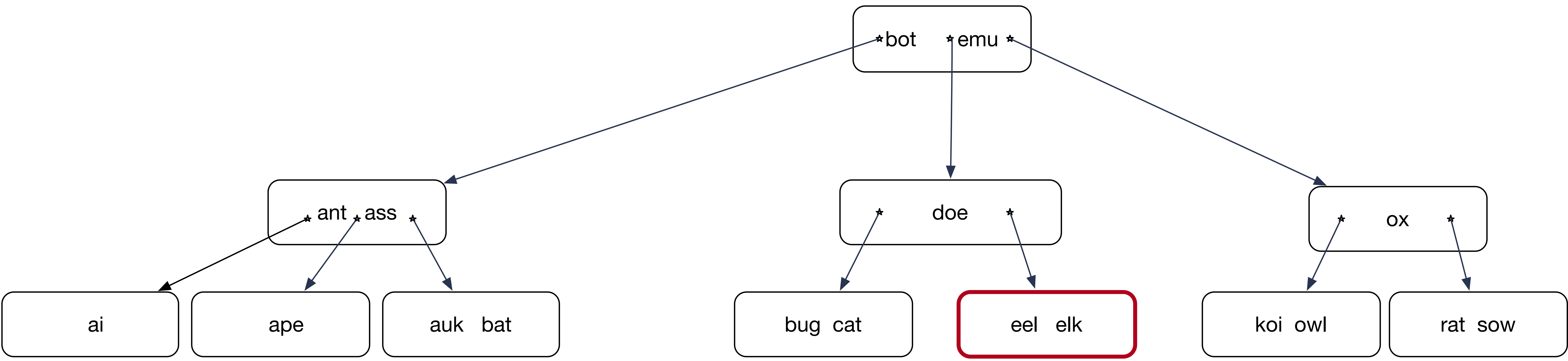
Cannot left-rotate: There is no right sibling
Cannot right-rotate: The left sibling has only one key
Need to merge: Combine the two nodes by bringing down “elk”

B-tree



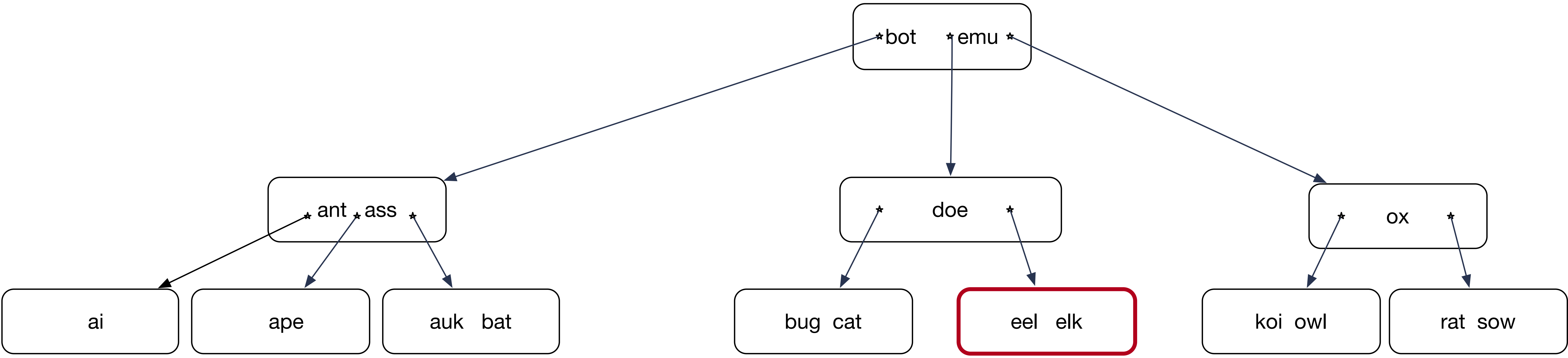
We can merge the two nodes because the number of keys combined is less than 2 *k*

B-tree



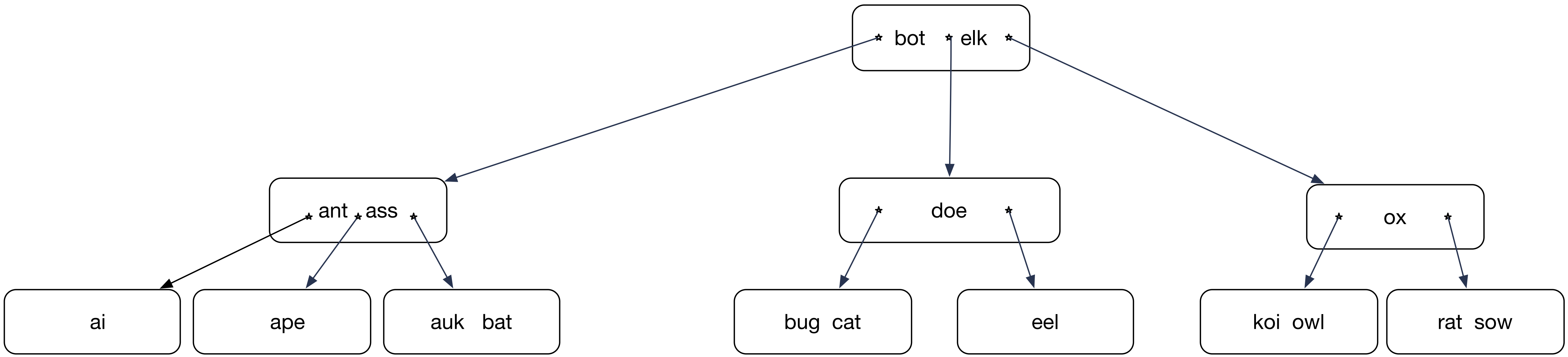
bot emu
ant ass
doe
ox
ai
ape
auk bat
bug cat
eel elk
koi owl
rat sow

B-tree



Delete “emu”

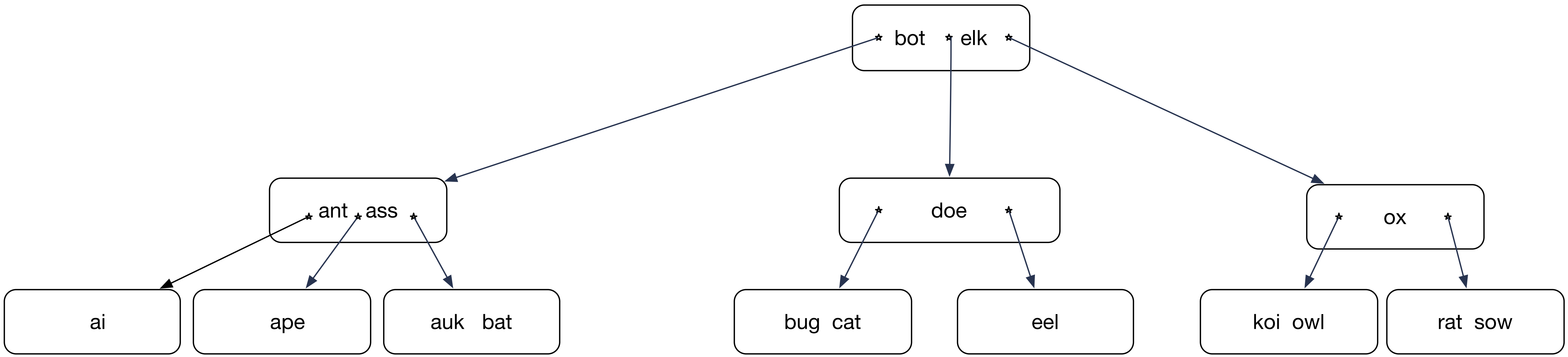
B-tree



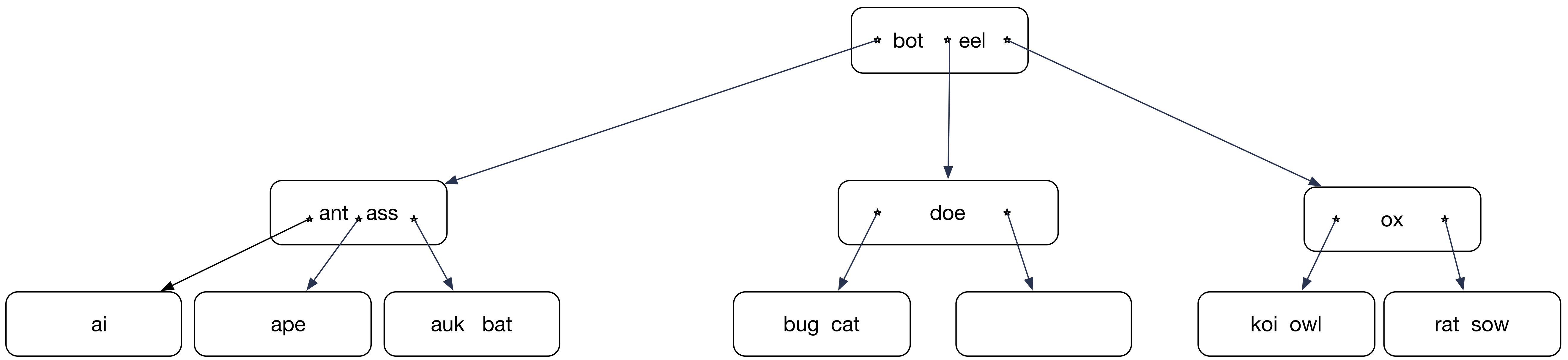
Switch predecessor, then delete from node

B-tree

Now delete “elk”

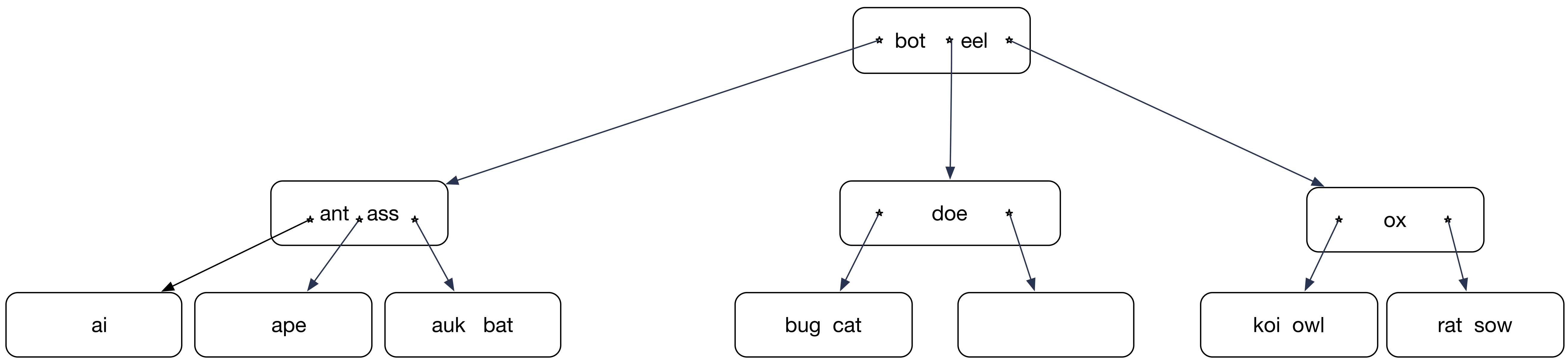


B-tree



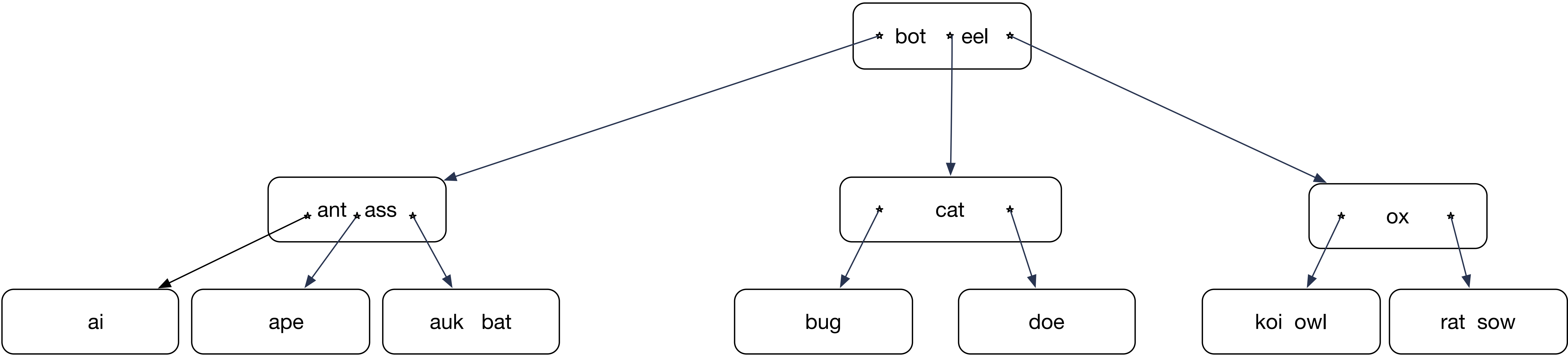
Results in an underflow

B-tree



Results in an underflow
But can rotate a key into the underflowing node

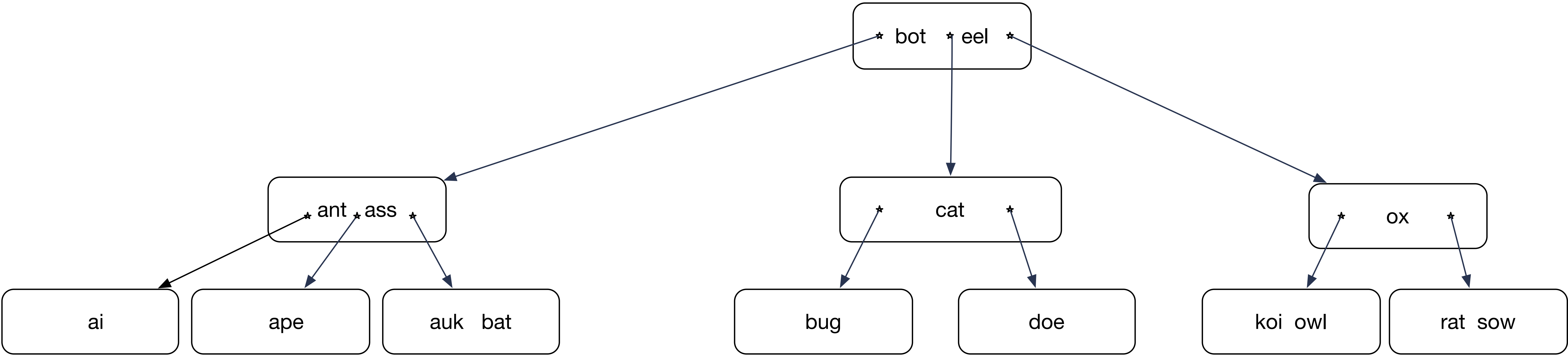
B-tree



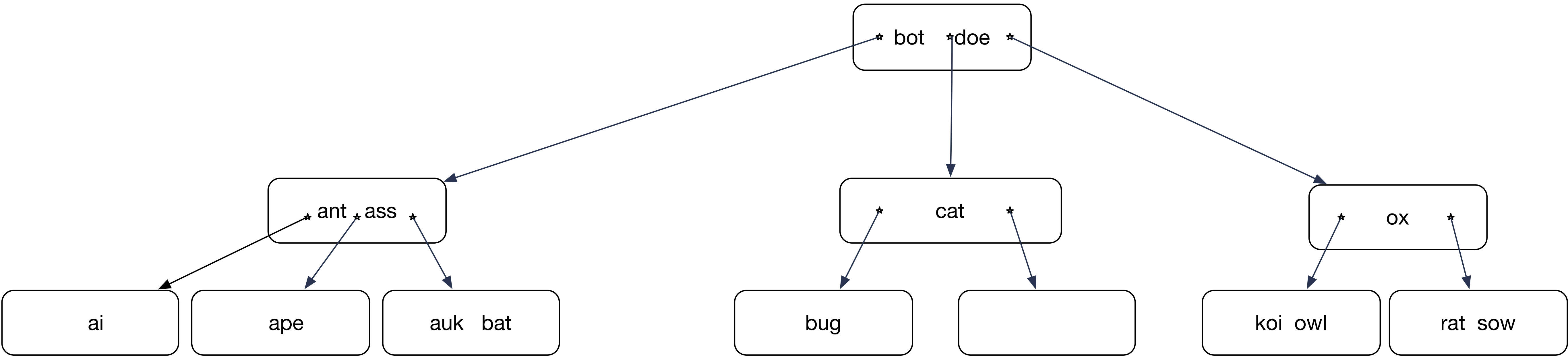
Result after right-rotation

B-tree

“Now delete “eel”

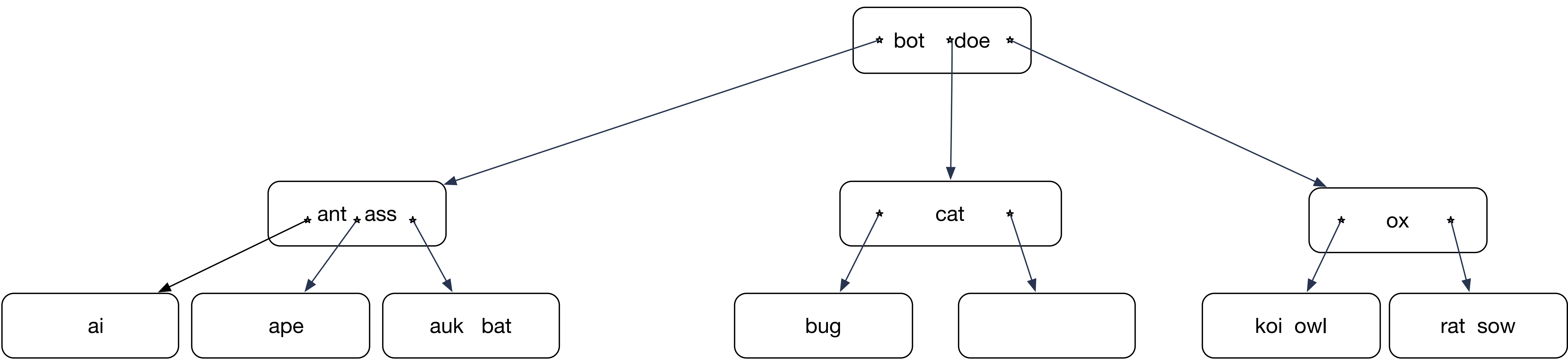


B-tree



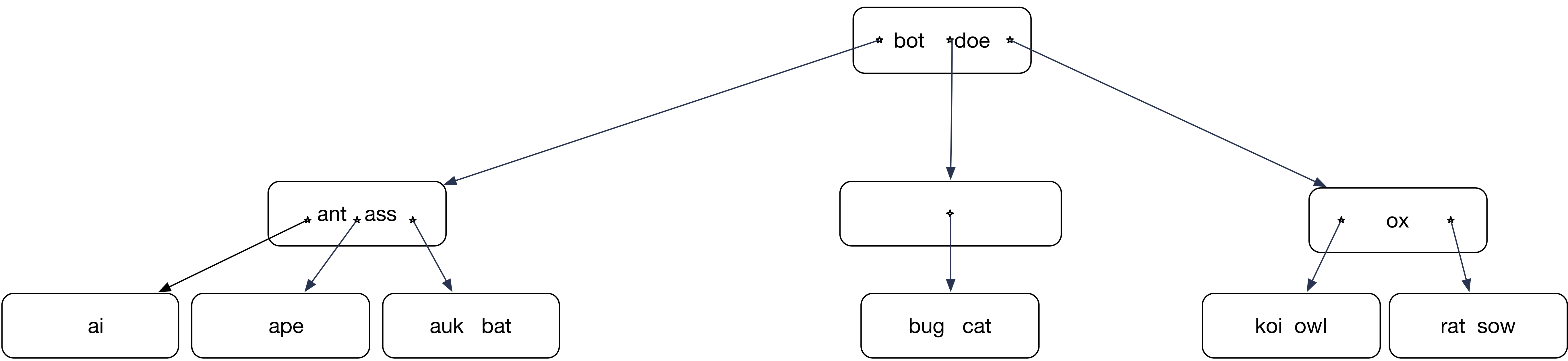
Interchange “eel” with its predecessor
Delete “eel” from leaf:
Underflow

B-tree



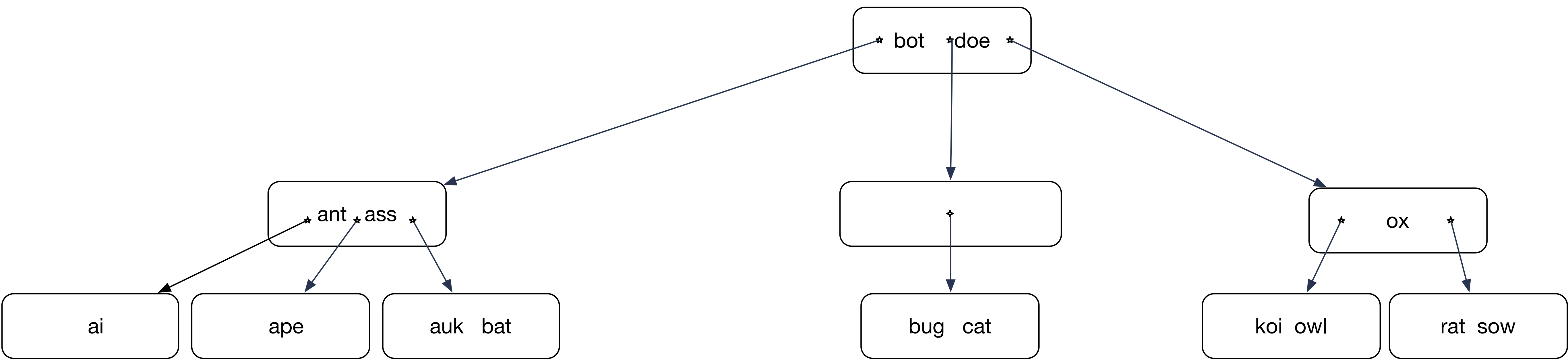
Need to merge

B-tree



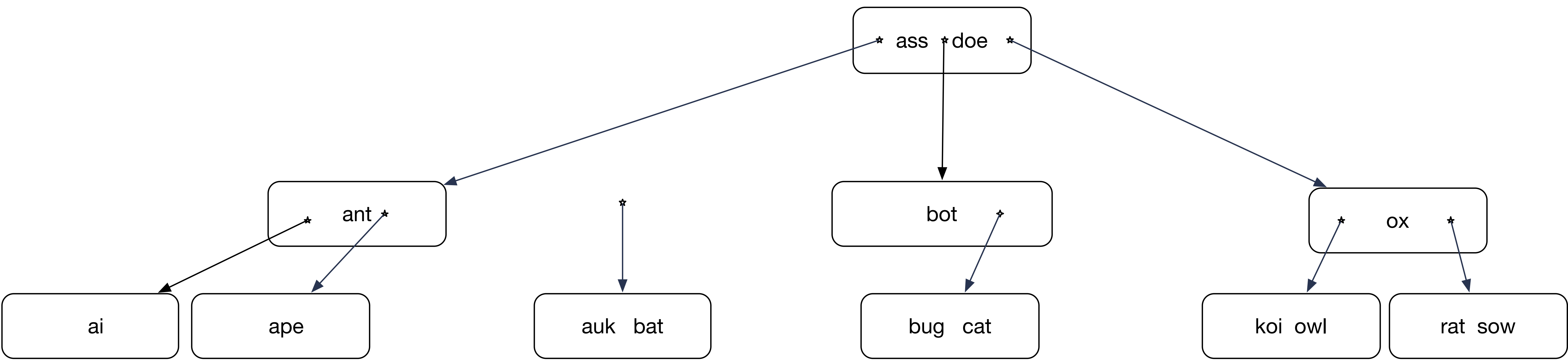
Merge results in another underflow
Use right rotate
(though merge with right sibling
is possible)

B-tree



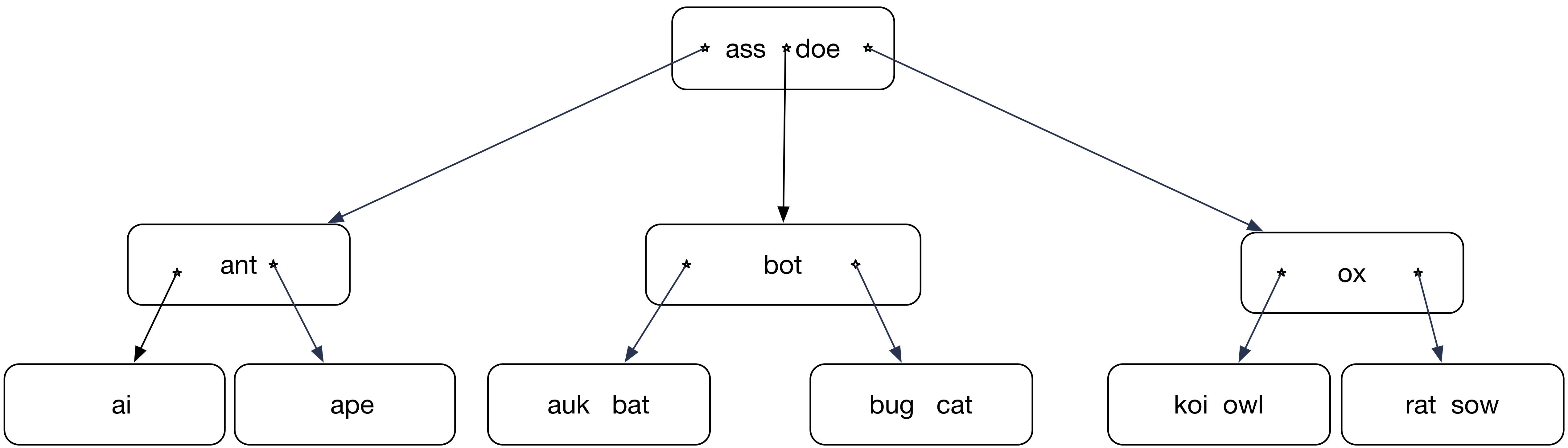
“ass” goes up, “bot” goes down
One node is reattached

B-tree



Reattach node

B-tree



In real life

- Use B+ tree for better access with block storage
  - Data pointers / data are only in the leaf nodes
  - Interior nodes only have keys as signals
  - Link leaf nodes for faster range queries.

In real life

- Storage systems:
    - Magnetic disk drives
        - Data stored in blocks of 4KB (originally 512B)
        - Access:
            - Seek + Rotate + Latency
            - ~5-15 msec
    - SSD:
        - Flash technology
        - Access:
            - ~1 msec
            - Unless using several channels

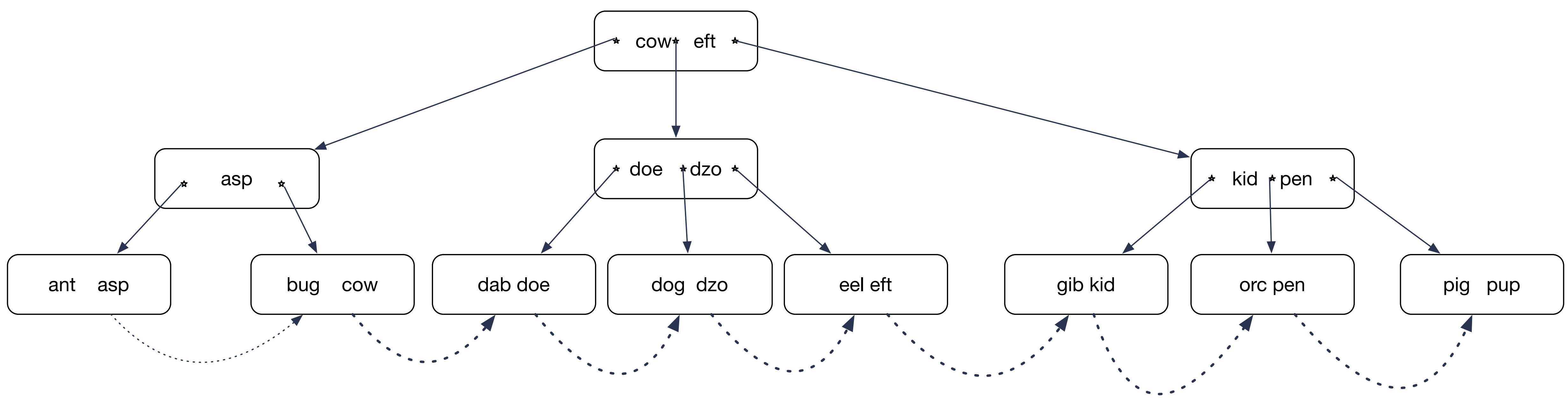
In Real Life

- Storage systems:
  - Transfer unit is a block / page
    - of size 4KB
  - We use DRAM as a cache
    - Store a node in a single page (or fixed-sized set of pages)
    - Only frequently used nodes should be in DRAM
      - This would be the upper layers of the hierarchy

In Real Life

- Best Strategy:
  - Treat interior nodes differently
    - because they are more frequently accessed
    - because most data is in the leaves
  - Do not store values, only keys in interior nodes
    - This way, each node contains the maximum amount of information
    - Which is used for navigation to the leaf

B+ Tree

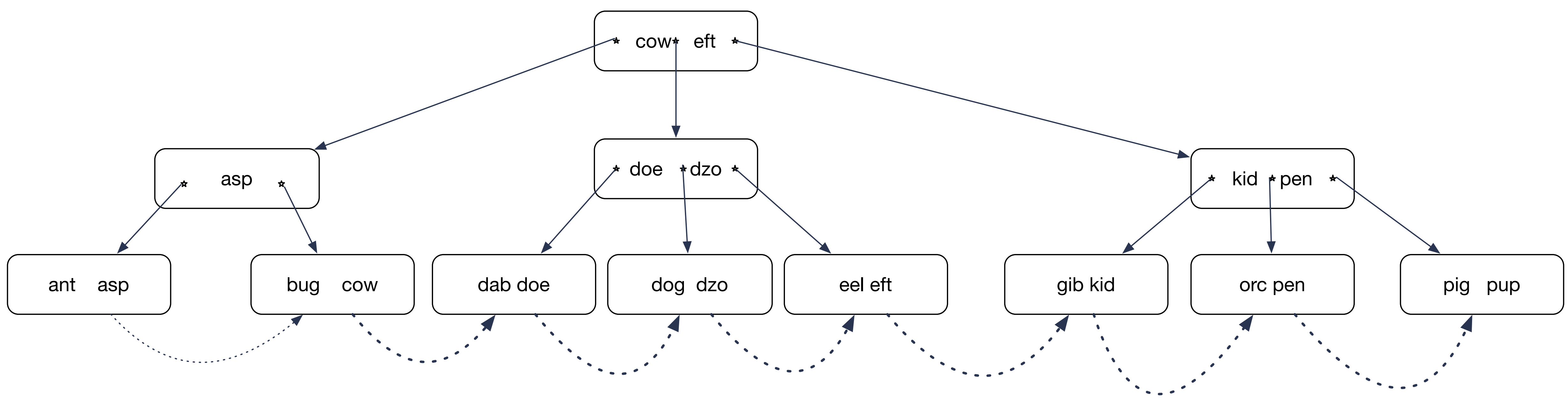


cow eft
asp
doe dzo
kid pen
ant asp
bug cow
dab doe
dog dzo
eel eft
gib kid
orc pen
pig pup

B+ Tree

- Interior nodes:
  - Contain only keys
  - The corresponding record is in a leaf, i.e. the key is repeated in the leaves

B+ Tree



cow eft
asp
doe dzo
kid pen
ant asp
bug cow
dab doe
dog dzo
eel eft
gib kid
orc pen
pig pup

B+ Tree

- Real life B+ trees:
  - Interior nodes have many more keys (e.g. 100)
  - Leaf nodes have as much data as they can keep
  - Need few levels:
    - Faster lookup

B+ Tree

- Range queries are easier:
  - Go to the first key in the range
  - Then follow the inter-node connections

- Size of node is an interesting optimization problem

B+ Trees

- Morale:
  - Data structures live in a concrete world
  - In Computer Science, underlying technology changes best practices
    - Therefore:
      - Computer Science is not a science, but an engineering discipline

B+ Trees

- Future memory / storage technologies
  - Non-volatile memory that combines
    - Speed of DRAM
    - Non-volatility of storage
    - Low cost of bytes of storage
    - Byte addressable
- Research question:
  - What B-tree is needed for these memories